Java Standards

[Sharmila Naveen]

[07/04/2010]

Table of Contents

1. Introduction

To build enterprise Java applications, which are reliable, scalable and maintainable, it is important for development teams to adopt proven design techniques and good coding standards. The adoption of coding standards results in code consistency, which makes it easier to understand, develop and maintain the application. In addition by being aware of and following the right coding techniques at a granular level, the programmer can make the code more efficient and performance effective.

2. Purpose and Scope

An effective mechanism of institutionalizing production of quality code is to develop programming standard and enforce the same through code reviews. This document delves into some fundamental Java programming techniques and provides a rich collection of coding practices to be followed by JAVA/J2EE based application development teams

The best practices are primarily targeted towards improvement in the readability and maintainability of code with keen attention to performance enhancements. By employing such practices the application development team can demonstrate their proficiency, profound knowledge and professionalism.

This document is written for professional Java software developers to help them:

- Write Java code that is easy to maintain and enhance
- Increase their productivity

3. File Organization

Java source are named as *. java while the compiled Java byte code is named as *.class file. Each Java source file contains a single public class or interface. Each class must be placed in a separate file. This also applies to non-public classes too.

If a file consists of sections, they should be separated by blank lines and an optional comment, identifying each section. Files longer than 2000 lines should be avoided.

Java classes should be packaged in a new java package for each self-contained project or group of related functionality. Preferably there should be an html document file in each directory briefly outlining the purpose and structure of the package. Java Source files should have the following ordering: Package and Import statements, beginning comments, Class and Interface Declarations. There should not be any duplicate import statement. There should not be any hard coded values in code. Max. No of Parameters in any class should be 12.

4. Source code style guidelines

4.1 Beginning Comments

All source files should begin with c-style header as follows carrying the Title, Version, Date in mm/dd/yy format and the copyright information.

```
/*
 * @(#)Title.java 2.12 04/05/02
 * Copyright (c) 2001-2002
 */
```

The header should be followed the package and import statements and then the documentation comments exactly in the following sequence and indented to the same level.

```
/**
 * Description
 * @author      <a href="mailto:author1@inetrrait.com">Author's Name</a>
* @version       1.0
 * @see           <a href="spec.html#section">Java Spec</a>
 * @since     since-text
 * @deprecated
 */
```

The tags see, since and deprecated are not mandatory and may be used when required.

4.2 Indentation

Four spaces should be used as the unit of indentation. The indentation pattern should be consistently followed throughout.

4.3 Left and Right braces

The starting brace should be at the end of the conditional and the ending brace must be on a separate line and aligned with the conditional. It is strongly recommended that all conditional constructs define a block of code for single lines of code.

4.4 Wrapping Lines

Lines longer than 80 characters should be avoided. When an expression will not fit on a single line, it should be broken according to these general principles:

- Break after a comma.
- Break after an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 4 spaces instead.

Example:

```
count = number.calculate(bytes, offset, length,
                    value, 0, estimatedCount);
long value = ((totalValue < plannedValue ) ?
                            totalValue : plannedValue );
```

4.5 White Space

Blank Lines

Blank lines improve readability by setting of sections of code that are logically related.

Two blank lines should be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods.
- Between the local variables in a method and its first statement.
- Before a block or single-line comment.
- Between logical sections inside a method to improve readability.
- Before and after comments.

4.6 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example: while (true)
- A blank space should appear after commas in argument lists.
- All binary operators except a period '.' should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.
- The expressions in a for statement should be separated by blank spaces.

Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank space.

```
myMethod((byte) Num, (Object) y);
myMethod((int) (c + 15), (int) (j + 4) );
```

However blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

4.7 Implementation Comments

Java codes should have implementation comments delimited by /*...*/ or //. For commenting out code a double slash i.e. // is recommended, while for multiple or single-line comments given as overview of code, the c-style comments i.e. /* */ should be used. For clarity in code, comments should be followed by a blank line. Code should have four styles of implementation comments as follows and anything that goes against a standard should always be document.

Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should be used at the beginning of each file and in places where code is to be explained. They can be used in places such as before or within methods. Block comments inside a method should be indented to the same level as the code they describe. A block comment should be preceded by a blank line to set it apart from the rest of the code. Block comments should be delimited by /*...*/.

During the process of development, the developer may need to leave some portions of the code to be reviewed and enhanced later. These portions should be specifically commented with a /* FIX ME */ tag specifying clearly the modifications to be made and the date of marking. This construct should however be sparingly used.

Single-Line & Trailing Comments

Short comments can appear on a single line indented to the level of the code that follows. A single-line comment may be preceded by a blank line if it gives clarity to code. It is recommended that single-line comments also be delimited by /*...*/.

Very short comments may appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. They may be used for explaining the use of a variable or any statement and should also be delimited by /*...*/.

Commenting Codes

The // comment delimiter can comment out a complete line or only a partial line. It should not be used on consecutive multiple lines for text comments, however, it may be used in single or consecutive multiple lines for commenting out sections of code.

Documentation Comment

Java codes should have documentation comments delimited by /**...*/. Documentation comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Simple Documentation comments should add to the clarity of code and should be followed by a blank line.

Classes and Interfaces

All comments using javadoc conventions should be shown. Each class should be documented describing the purpose of the class, guaranteed invariants, usage instructions, and/or usage examples. Also including any reminders or disclaimers about required or desired improvements using the tags mentioned in Section 3.3.1

Variables

First the static variables should be declared in the sequence public also all the static variables defined before the methods in the classes, then protected, then package level and then the private followed by instance variables in the same sequence. Nature, purpose, constraints, and usage of static and instances variables should be documented using javadoc conventions. A blank line should follow the declarations.

```
/** Document the variable
 *  @see          ClassName#method()
*/
```

- **OPERATORS**

The parenthesis should be effectively used to avoid operator precedence.

4.8 Methods

Each method should declare the javadoc tags exactly in the sequence as given below. Each line item begins with an asterisk. All subsequent lines in multiline component are to be indented so that they line up vertically with the previous line. For reference, the javadoc tags are explained in detail in Annexure.
Example

```
/**
 * Description:
 * @param       <Mandatory Tag> for description of each parameter
 * @return      <Mandatory Tag> except for constructor and void>
 * @exception   <Optional Tag>
 * @see                 <Optional Tag>
 * @since       <Optional Tag>
 * @deprecated          <Optional Tag>
 */
```

Description

Every method should include a header at the top of the source code that documents all of the information that is critical to understanding it.

Detailed description of the method may include the intent of method i.e. what and why the method does, what a method should be passed as parameters and what it returns, any Known bugs, exceptions that the method throws, information on visibility decisions., how a method changes the object, pre and post conditions, side effects , dependencies , implementation notes , who should be calling this method , whether the method should or should not be overridden , where to invoke super when overriding , control flow or state dependencies that need to exist before calling this method.

PARAMETER SECTION

Describes the type, class, or protocol of all the method or routine arguments. Should describe the parameters intended use and constraints. Every function parameter value should be checked before being used (Usually check for nulls and Data Validation).

EXAMPLE:

```
* @param aSource the input source string which cannot be 0-length.
```

RETURNS SECTION

This section is used to describe the method return type. Specifically, it needs to detail the actual data type returned, the range of possible return values, and where applicable, error information returned by the method. Every function should return the correct value at every function return point or throw correct Exceptions in case of Errors.

Example:

```
* @return Possible values are 1..n.
```

EXCEPTION SECTION

The purpose section is a complete description of all the non-system exceptions that this method throws. A description about whether the exception is recoverable or not should also be included. If applicable, a recovery strategy for the exception can be described here.

```
* @exception ResourceNotFoundException. recoverable.
```

Annexure explains the commonly used tags for javadoc.

In addition to the method documentation, it is also required to include comments within the methods to make it easier to understand, maintain, and enhance.

- The control structures i.e. what each control structure, such as loop, does should be documented.
- Any assumptions or calculations should be documented.
- Each local variable should have an end line comment describing its use.
- Any complex code in a method should be documented if it is not obvious.
- If there are statements that must be executed in a defined order then this fact should be documented.

5.0 Declarations

One declaration per line is recommended since it encourages commenting and enhances the clarity of code. The order and position of declaration should be as follows:

First the static/class variables should be placed in the sequence: First public class variables, protected, package/default level i.e. with no access modifier and then the private. As far as possible static or class fields should be explicitly instantiated by use of static initializers because instances of a class may sometimes not be created before a static field is accessed.

- Instance variables should be placed in the sequence: First public instance variables, protected, package level with no access modifier and then private.
- Next the class constructors should be declared.
- This should be followed by the inner classes, if applicable
- Class methods should be grouped by functionality rather than by scope or accessibility to make reading and understanding the code easier.
- Declarations for local variables should be only at the beginning of blocks e.g. at the beginning of a try-catch construct.

- Local declarations that hide declarations at higher levels should be avoided. For example, same variable name in an inner block.
- Numerical constants should not be coded directly except 1, 0, -1.

6.0 Standards for Statements

Each line should contain at most one statement. While compound statements are statements that contain lists of statements enclosed in braces. The enclosed statements should be indented one more level than the compound statement. The opening brace should be at the end of the line that begins the compound statement. The closing brace should begin a line and be indented to the beginning of the compound statement. Braces should be used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. A boolean expression / function should only be compared to a boolean constants. goto statements should be avoided.

Try to move invariable computation outside the loop.

E.g. D += 9*24*pie *x

Where pie is a constant then 9*24*pie can be moved out and assigned to a local variable and used inside a loop where x changes in the loop

- return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way.

Example:

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

- if, if-else Statements

The 'if' keyword and conditional expression must be placed on the same line. The if statements must always use braces {}

Example:

```
if (expression) {
   statement;
} else {
   statement;
}
```

- for Statements

A for statement should have the following form:

```
for ( int i = 0; i < 10; i++ ) {
   System.out.println(i);
}
```

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause) may be used. The keyword 'for' and the parenthesis should be separated by a space and the expressions in a for statement should be separated by blank space. The statement block is placed on the next line. The closing curly brace starts in a new line, indented to match its corresponding opening statement.

- while Statements

The 'while' construct uses the same layout format as the 'if' construct. The 'while' keyword should appear on its own line, immediately followed by the conditional expression. The keyword 'while' and the parenthesis should be separated by a space. The statement block is placed on the next line. The closing curly brace starts in a new line, indented to match its corresponding opening statement.

Example:

```
while (expression) {
   statement;
}
```

- do-while Statements

The DO..WHILE form of the while construct should appear as shown below:

Examples:

```
do {
  statement;
} while (expression);
```

The statement block is placed on the next line. The closing curly brace starts in a new line, indented to match its corresponding opening statement.

- switch Statements

The 'switch' construct should use the same layout format as the 'if' construct. The 'switch' keyword should appear on its own line, immediately followed by its test expression. The keyword 'switch' and the parenthesis should be separated by a space.

The statement block should be placed on the next line. The closing curly brace starts in a new line, indented to match its corresponding opening statement. Every time a case falls through, i.e. does not include a break statement, a comment should be added where the break statement would normally be. Every switch statement must include a default case. A break in the default case is redundant, but it prevents error if another case is added later and therefore may be added. All missing switch case break statements should be correct and marked with a comment.

Examples:

```
switch (expression) {
  case n:
    statement;
    break;
  case x:
    statement;
                    /* Continue to default case */
  default:                  /* always add the default case */
```

```
    statement;
    break;
}
```

- try-catch Statements

In the try/catch construct the 'try' keyword should be followed by the open brace in its own line. This is followed by the statement body and the close brace on its own line. This may follow any number of 'catch' phrases - consisting of the 'catch' keyword and the exception expression on its own line with the 'catch' body; followed by the close brace on its own line. Try-catch should be accomplish with finally block to destroys all Objects not required. There should not be any empty try-catch blocks.

Example:

```
try {
   statement;
} catch (ExceptionClass e) {
   statement;
} finally {
   statement;
}
```

- **Naming Conventions**

Naming conventions make programs more understandable by making them easier to read. Following conventions should be followed while naming a class or a member:

- Use full English descriptors that accurately describe the variable, method or class. For example, use of names like totalSales, currentDate instead of names like x1, y1, or fn.
- Terminology applicable to the domain should be used. Implying that if user refers to clients as customers, then the term Customer should be used for the class, not Client.
- Mixed case should be used to make names readable with lower case letters in general capitalizing the first letter of class names and interface names.
- Abbreviations should not be used as far as possible, but if used, should be documented and consistently used.

- Very long or similar names for classes or methods should be avoided.
- Standard acronyms such as SQL should also be represented in mixed case as sqlDatabase().

7.0 Standards for Methods

Naming Methods

Methods should be named using a full English description, using mixed case with the first letter of any non-initial word capitalized. It is also common practice for the first word of a method name to be a strong, active verb. e.g. getValue(), printData(), save() ,delete(). This convention results in methods whose purpose can often be determined just by looking at its name. It is recommended that accessor methods be used to improve the maintainability of classes.

Getters

Getters are methods that return the value of a field. The word 'get' should be prefixed to the name of the field, unless it is a boolean field where 'is' should be prefixed to the name of the field . e.g. getTotalSales(), isPersistent(). Alternately the prefix 'has' or 'can' instead of 'is' for boolean getters may be used. For example, getter names such as hasDependents() and canPrint() can be created. Getters should always be made protected, so that only subclasses can access the fields except when an 'outside class' needs to access the field when the getter method may be made public and the setter protected.

Setters

Setters, also known as mutators, are methods that modify the values of a field. The word 'set' should be prefixed to the name of the field for such methods type. Example: setTotalSales(), setPersistent(boolean isPersistent)

Getters for Constants

Constant values may need to be changed over a period of time. Therefore constants should be implemented as getter methods. By using accessors for constants there is only one source to retrieve the value. This increases the maintainability of system.

Accessors for Collections

The main purpose of accessors is to encapsulate the access to fields. Collections, such as arrays and vectors need to have getter and setter method and as it is possible to add and remove to and from collections, accessor methods need to be included to do so. The advantage of this approach is that the collection is fully encapsulated, allowing changes later like replacing it with another structure, like a linked list.

Examples: getOrderItems(), setOrderItems(), insertOrderItem(), deleteOrderItem(), newOrderItem()

Method Visibility

For a good design, the rule is to be as restrictive as possible when setting the visibility of a method. If a method doesn't have to be private then it should have default access modifier, if it doesn' t have to be default then it should be made protected and if it doesn' t have to protected only then it should be made public. Wherever a method is made more visible it should be documented why.

Access modifier for methods should be explicitly mentioned in cases like interfaces where the default permissible access modifier is public.

Standards for Parameters (Arguments) To Methods

Parameters should be named following the same conventions as for local variables. Parameters to a method are documented in the header documentation for the method using the javadoc@param tag.

However:

- Cascading method calls like method1().method2() should be avoided.
- Overloading methods on argument type should be avoided.
- It should be declared when a class or method is thread-safe.
- Synchronized methods should be preferred over synchronized blocks.
- The fact that a method invokes wait should always be documented
- Abstract methods should be preferred in base classes over those with default no-op implementations.
- All possible overflow or underflow conditions should be checked for a computation.

There should be no space between a method/constructor name and the parenthesis but there should be a blank space after commas in argument lists.

```
public void doSomething( String firstString, String secondString ) {

}
```

8. 0 Naming Convention standards

Naming Variables

Use a full English descriptor for variable names to make it obvious what the field represents. Fields, that are collections, such as arrays or vectors, should be given names that are plural to indicate that they represent multiple values. Variable names should not start with an underscore _ or dollar sign $ characters and should be short and meaningful. The choice of a variable name should be mnemonic i.e., designed to indicate to the casual observer the intent of its use. Single character variable names should be avoided except for temporary "throwaway" variables.

Naming Components

For names of components, full English descriptor should be used, post fixed by the Component type. This makes it easy to identify the purpose of the component as well as its type, making it easier to find each component in a list. Therefore names like NewHelpMenuItem, CloseButton should be preferred over Button1, Button2, etc.

Naming Constants

Constants, whose values that do not change, are typically implemented as static final fields of classes. They should be represented with full English words, all in uppercase, with underscores between the words like FINAL_VALUE.

Naming Collections

A collection, such as an array or a vector, should be given a pluralized name representing the types of objects stored by the array. The name should be a full English descriptor with the first letter of all non-initial words capitalized like customers, orderItems, aliases

Naming Local Variables

In general, local variables are named following the same conventions as used for fields, in other words use of full English descriptors with the first letter of any non-initial word in uppercase. For the sake of convenience, however, this naming convention is relaxed for several specific types of local variable like Streams, Loop counters, Exceptions.

Name hiding or data hiding refers to the practice of naming a local variable, argument, or methods the same or similar as that of another one of greater scope in same or super class. This may lead to confusion and should be avoided.

Naming Streams

When there is a single input and/or output stream being opened, used, and then closed within a method the common convention is to use 'in' and 'out' for the names of these streams, respectively.

Naming Loop Counters

Loop counters are a very common use for local variables therefore the use of i, j, or k, is acceptable for loop counters where they are obvious. However, if these names are used for loop counters, they should be used consistently. For complex nested loops the counters should be given full meaningful English descriptors.

9.0 Variable Assignments

- Assigning several variables to the same value in a single statement should be avoided, i.e., we should avoid constructs like var1 = var2 = var3 = 0;
- Assignment operator should not be used in a place where it can be easily confused with the equality operator.
- Embedded assignments in an attempt to improve run-time performance should be avoided

- **Variable Visibility**

For reasons of encapsulation, fields should not be declared public. All fields should be declared private unless necessary otherwise. Fields should never be accessed directly, instead accessor methods should be used i.e. private members should be accessed through methods. All fields should be declared private and accessed by getter and setter methods also called accessors.

- **Documenting & Declaring a Variable**

Every field should be documented well enough so that other developers can understand it. There is a need to document description, all applicable invariants, visibility decisions. Wherever a field is made more

visible it should be documented why. There are some conventions regarding the declaration and documentation of local variable. These conventions are:

- Declare one local variable per line of code.
- Document local variables with an end line comment.
- Declare local variables at the top of the block of code.
- Use local variables for one thing only at a time.
- However the following should be taken into consideration while using variables:
- Instance variables should not be declared public.
- Implicit initializers for static or instance variables should not be relied and initialization should be explicit.
- Use of static should be minimized as they act like globals. They make methods more context-dependent, hide possible side effects, sometimes present synchronized access problems and are the source of fragile, non-extensible constructions. Also, neither static variables nor methods are overridable in any useful sense in subclasses.
- Generally prefer double to float and use int for compatibility with standard Java constructs and classes.
- Use final and/or comment conventions to indicate whether instance variables that never have their values changed after construction are intended to be constant (immutable) for the lifetime of the object.
- Declare and initialize a new local variable rather than reusing/reassigning an existing one whose value happens to no longer be used at that program point.
- Assign null to any reference variable that is no longer being used to enable garbage collection.
- Same names of variables or methods should be avoided in methods and subclasses.

10.0 Standards for Classes, Interfaces, Packages, and Compilation Units

10.1 Standards for Classes

Naming Classes

Class names should be simple full English descriptor nouns, in mixed case starting with the first letter capitalized and the first letter of each internal word also capitalized. Whole words should be used instead of acronyms and abbreviations unless the abbreviation is more widely used than the long form, such as URL or HTML.

Class Visibility

Package or default visibility may be used for classes internal to a component while public visibility may be used for other components. However, for a good design, the rule is to be as restrictive as possible when setting the visibility. The reason why the class is public should be documented. Each class should have an appropriate constructor.

Documenting a Class

The documentation comments for a class start with the header for class with filename, version, copyright and related information. The documentation comments should precede the definition of a class and should contain necessary information about the purpose of the class, details of any known bugs, examples etc. as illustrated in. The development/maintenance history of the class should be entered as comments in the configuration management tool at the time of baselining the source code and in the file header as well.

10.2 Standards for Interfaces

The Java convention is to name interfaces using mixed case with the first letter of each word capitalized like classes. The preferred convention for the name of an interface is to use a descriptive adjective, such as Runnable or Cloneable. Interfaces should be documented specifying the purpose of the interface and how it should and shouldn't be used. Method declarations in interfaces should explicitly declare the methods as public for clarity.

10.3 Standards for Packages

10.3.1 Naming Packages

The rules associated with the naming of packages are as follows:

Unless required otherwise, a package name should include the organization's domain name, with the top-level domain type in lower case ASCII letters i.e. com.<Name of company> followed by project name and sub project name as specified in ISO Standard 3166, 1981.

Subsequent components of the package name vary according to requirements.

Package names should preferably be singular.

10.3.2 Documenting a Package

There should be one or more external documents in html format with the package name that describe the purpose of the packages documenting the rationale for the package, the list of classes and interfaces in the package with a brief description of each so that other developers know what the package contains.

11.0 Configuration Management

While controlling the source code in configuration management tool the development/maintenance history of the class should be entered as comments in the configuration management tool. The comments should include details like name of the java file and package, name of method/s changed/ modified/ added / deleted, brief description of changes, name of author/s or modifier/s and reference of any known or fixed bugs.

12.0 Best Practices

Efficient String Concatenation:-

For making a long string by adding different small strings always use append method of java.lang.StringBuffer and never use ordinary '+' operator for adding up strings.

Optimal Use Of Garbage Collector: -

For easing out the work of java Garbage Collector always set all referenced variables used to 'null' explicitly thus de-referencing the object which is no more required by the application and allowing Garbage Collector to swap away that variable thus realizing memory.

Writing Oracle Stored Procedures Optimally:-

Avoid using 'IN' and 'NOT IN' clause and rather try to use 'OR' operator for increasing the response time for the Oracle Stored Procedures.

Using variables in any Code Optimally:-

Try to make minimum number of variables in JSP/Java class and try to use already made variables in different algorithms shared in same JSP/Java class by setting already populated variable to 'null' and then again populating that variable with new value and then reusing them.

Try to write minimum java code in JSPs:-

Big patches of java code in JSP should be avoided and is should be rather shifted to some Wrapper/Helper/Bean class which should be used together with every JSP. Number of method calls from JSP should be reduced as much as possible to achieve maximum efficiency and the least response time. In brief JSP should be designed as light as possible.

Try to reduce the number of hits to database:-

Number of hits to the database should be reduced to minimum by getting data in a well arranged pattern in minimum number of hits by making the best use of joins in the database query itself rather than getting dispersed data in more number of hits.

Note:- If the data is very large then this approach should be avoided since it slow down the loading of the JSP page itself on the client.

Caching of EJB References:-

To avoid costly lookup every time any remote object is required, its better to cache the home reference and reusing it.

Note:- With time the cached home reference may get stale so proper care should be take to do a re-lookup as soon as such a situation arises.

Heavy Objects should not be stored in Session of JSPs:-

Storing heavy objects in the Session can lead to slowing of the running of the JSP page so such case should be avoided.

Always Use JDBC connection Pooling:-

Always use javax.sql.DataSource which is obtained through a JNDI naming lookup. Avoid the overhead of acquiring a javax.sql.DataSource for each SQL access. This is an expensive operation that will severely impact the performance and scalability of the application.

Release HttpSessions when finished:-

Abandoned HttpSessions can be quite high.HttpSession objects live inside the engine until the application explicitly and programmatically releases it using the API,

javax.servlet.http.HttpSession.invalidate (); quite often, programmatic invalidation is part of an application logout function.

Release JDBC resources when done:-

Failing to close and release JDBC connections can cause other users to experience long waits for connections. Although a JDBC connection that is left unclosed will be reaped and returned by Application Server after a timeout period, others may have to wait for this to occur. Close JDBC statements when you are through with them. JDBC ResultSets can be explicitly closed as well. If not explicitly closed, ResultsSets are released when their associated statements are closed. Ensure that your code is structured to close and release JDBC resources in all cases, even in exception and error conditions.

Minimize use of System.out.println:-

Because it seems harmless, this commonly used application development legacy is overlooked for the performance problem it really is. Because System.out.println statements and similar constructs synchronize processing for the duration of disk I/O,they can significantly slow throughput.

Minimum use of java.util.Vector:-

Since most of the commonly used methods in Vector class are Synchronized which makes any method call or any variable heavy as compared to those which are not synchronized so it's a better practice to use any other Collection sibling whose methods are not synchronized for eg. java.util.ArrayList.

Vector (and other "classic" utility classes such as Hashtable) are synchronized on all methods. This means that even you do not access the Vector from multiple threads you pay a performance penalty for thread safety. ArrayList is not synchronized by default - although the collections framework provides wrappers to provide thread safety when needed. Which accounts for the memory usage to be very high if they are holding heavy amount of data.

Synchronization has a cost

- Putting synchronized all over the place does not ensure thread safety.
- Putting synchronized all over the place is likely to deadlock.
- Putting synchronized all over will slow your code and prevent it from running when it should. This accounts for memory leaks.

Using JS files and other include files in Jar:-

When we are developing web based large number of include file. All includes file must be put in jar and accessed which is very fast. The method to do that we write a property class and get all the properties from the jar.

Forgotten references keep Objects alive.

Be careful with indexes and caches.

Hashtable h = new Hashtable();

Object[] storage = new Object[1000];

h.put(new Integer(1), storage);

h.remove(storage);

storage = null

This accounts for the Memory Leaks

Only consider pooling "heavy" objects:

Large Arrays

Images

Threads

With newer Java VMs (i.e. Java HotSpot) the cost of maintaining a pool can outweigh allocation and collection.

References

- Code Conventions for the JavaTM Programming Language (http://java.sun.com/docs/codeconv/)
- Java Programming Style Guidelines (http://geosoft.no/javastyle.html)
- "The Java Book " by Prateek Khanna
- AmbySoft Inc. Coding Standards for Java v17.01d (http://www.ambysoft.com/javaCodingStandards.html)
- ChiMu OO and Java Development: Guidelines and Resources v 1.8 (www.chimu.com)
- http://java.sun.com/j2se/javadoc/writingdoccomments/

Annexure 2 : Code Example

```
package com.project1;                    /* package should follow the naming conventions */
import  com.project2.Class1;     /* Import specific class and not all.*/
/* Leave 2 lines between sections of a source file*/
```

```
/**
 *
 * This is an example class. This class creates an object of <CODE>MyExample
 * </CODE> class. Detailed description, documentation and  code can come here.
 * If required, an image can be placed here using the img tag in HTML as follows:
 * <p><img src="images/Sample.gif"><p> HTML tags like hyperlinking, table etc.
 * may be used here, if required. Use code style for Java keywords,  package
 * names, class names, method names, interface names, field names, argument
 * names, code examples. However formatting tags like H1, H2 should not be used
 * as javadoc has a consistent format for generating an API. Use the @link tag
 * to link to another class like {@link com.Sample.MyExample}.
 * Code sample may be shown using the blockquote tag as follows
 * <p><blockquote><pre>
import java.awt.*;
 * import java.applet.Applet;
 * </pre></blockquote>
 *
 * @author                  <a href = "mailto:first@inetrrait.com">First Author</a>
 * @author                  <a href = "mailto:second@inetrrait.com">Second Author</a>
 * @version                 2.11      04 May 2002
 * @see                     <a href="index.html#Section">Java Spec</a>
 * @see                     com.Sample.MyExample MyExample
 * @since                   Ver 1.0
 * @serial          or @serialField or @serialData)
 * @deprecated      As of ver 1.0
 */
```

```
public class MyExample  {

  /** The documentation for public static variable instanceCounter
   * @see                          MyExample#calculate()
   */
  public static int instanceCount = 0;              /* Declare public static variable first */

  /**
   * The documentation for private static variable classVar2 that happens to be
   * more than one line long
   */

  private static Object classVar2;          /* Declare public, protected, package &
                                                        then private static variables.*/

  /** The documentation for public instance variable instanceVar1 */
  public Project instanceVar1;              /* Declare public instance variable first*/

  /**  The documentation for protected instance variable instanceVar2 */
  protected int instanceVar2;

  /**  The documentation for private instance variable instanceVar3 */
  private boolean[] instanceVar3;                   /*        One declaration per line.*/

  /**
```

```
* Default constructor which creates an empty object of MyExample class.
   */
  public MyExample() {
     instanceCount++;                                  /*implementation for constructor goes here*/
  }

  /**
   * gets the value of n factorial by performing the operation n! = 1*2*3* ...*n
   * @param  int The number whose factorial is to be calculated.
   * @return int
   * @see         com.Sample.MyExample#calculate()
   * @since Ver 1.0
   * @deprecated As of Ver 1, replaced by {@link com.Sample.MyExample#calculate()}
   */

  public int calculate(int number) {
     int value = 1;                    /*       Declare all variables at the top in the sequence public,
                                                 protected, default, private, leaving blank line after declarations.*/
     if (number > 0) {          /* if statements should always use braces */
                 for (int outer = 1; outer < (number + 1); outer++) {
      /*  Short comment on a single line indented to the level of the code */
                value = value * outer;
                      }
                      return value;
                   } else {
                      return 0;                /* Short comments can appear in line of code */
                   }                                 /* shifted far enough to separate from statements. */
```

```
            //if (number < 0 ) {
            //}                                   Double slash can be used for commenting out sections
            //else {of code or a single line but not for explanatory
            //    return 0;          comments where /* */ should be used.
            //}
          }
}
```

13.0 Java Documentation Tags

Tag	Used for	Purpose
@author name	Interfaces, Classes, Interfaces	Indicates the author(s) of a given piece of code. One tag per author should be used.
@deprecated	Interfaces, Classes, Member, Functions	Indicates that the API for the class has been deprecated and therefore should not be used any more.
@exception name description	Member, Functions	Describes the exceptions that a member function throws. You should use one tag per exception and give the full class name for the exception.
@param name description	Member, Functions	Used to describe a parameter passed to a member, function, including its type/class and its usage. Use one tag per parameter.
@return description	Member, Functions	Describes the return value, if any, of a member function. You should indicate the type/class and the potential use(s) of the return value.
@since	Interfaces, Classes, Member, Functions	Indicates how long the item has existed, i.e. since JDK 1.1
@see ClassName	Classes, Interfaces, Member, Functions, Fields	Generates a hypertext link in the documentation to the specified class. You can, and probably should, use a fully qualified class name.

Tag	Used for	Purpose
@see ClassName# functionName	Classes, Interfaces, Member, Functions, Fields	Generates a hypertext link in the documentation to the specified member function. You can, and probably should, use a fully qualified class name.
@version text	Classes, Interfaces	Indicates the version information for a given piece of code.

Annexure A: Technical points

- Apart from the standards mentioned already following should be considered while writing java code:
- Instance /class variables should not be made public as far as possible.
- A constructor or method must explicitly declare all unchecked (i.e. runtime) exceptions it expects to throw. The caller can use this documentation to provide the proper arguments.
- Unchecked exceptions should not be used instead of code that checks for an exceptional condition. e.g. Comparing an index with the length of an array is faster to execute and better documented than catching ArrayOutOfBoundsException.
- If Object.equals is overridden, also override Object.hashCode, and vice-versa.
- Override readObject and writeObject if a Serializable class relies on any state that could differ across processes,including,in particular,hashCodes and transient fields.
- If clone() may be called in a class, then it should be explicitly defined, and declare the class as implements Cloneable.
- Always use method equals instead of operator == when comparing objects. In particular, do not use == to compare Strings unless comparing memory locations.
- Always embed wait statements in while loops that re-wait if the condition being waited for does not hold.
- Use notifyAll instead of notify or resume when you do not know exactly the number of threads which are waiting for something

- Embed casts in conditionals. This forces to consider what to do if the object is not an instance of the intended class rather than just generating a ClassCastException. For example:

```
C cx = null;
if (x instanceof C) {
cx = (C) x
} else {
doSomething();
}
```

- When throwing an exception, do not refer to the name of the method which has thrown it but specify instead some explanatory text.
- Document fragile constructions that have been used solely for the sake of optimization.
- Document cases where the return value of a called method is ignored.
- Minimize * forms of import Be precise about what you are importing.
- Prefer declaring arrays as Type[] arrayName rather than Type arrayName[].
- StringBuffer should be preferred for cases involving String concatenations. Wherever required String objects should be preferably created with a new and not with the help of assignment, unless intentionally as they remain in the String pool even after reference is nullified.
- All class variables must be initialized with null at the point of declaration.
- All references to objects should be explicitly assigned 'null' when no more in use to make the objects available for garbage collection.
- As far as possible static or class fields should be explicitly instantiated by use of static initializers because instances of a class may sometimes not be created before a static field is accessed.
- Minimize statics (except for static final constants).
- Minimize direct internal access to instance variables inside methods.
- Declare all public methods as synchronized.
- Always document the fact that a method invokes wait.

- Classes designed should be easily extensible. This will be very important in the event that the currently designed project needs to be enhanced at a later stage.
- It is very important to have the finally clause (whenever required) because its absence can cause memory leakage and open connections in a software.